Capítulo 1

Java® Platform

Este capítulo establece los conocimientos necesarios para iniciar a escribir aplicaciones en el lenguaje de programación Java®. Las secciones tratan una introducción, técnica teórica de los fundamentos, materiales, montaje, primera compilación y estructura léxica.

1.1. Fundamentos

1.1.1. Historia

En diciembre de 1990 Sun Microsystems patrocinó un proyecto interno de investigación denominado Green cuyo objetivo fue anticipar y planificar la *próxima ola* en informática. En el verano de 1992 el equipo Green presento el *7 (*StarSeven*), un tipo de PDA que tenía un *agente* (una entidad de software que realiza tareas en nombre del usuario) llamado Duke (la mascota de la tecnología Java, inventado y dibujado por Joe Palrang) además ejecutaba bajo un lenguaje propio e independiente del procesador. El lenguaje fue creado en 1991 [1, p. 27] por James Gosling y la llamó Oak, en honor al árbol fuera de su ventana. *7 se terminó y demostró el 3 de septiembre de 1992.[2]
A medida que el proyecto ganó impulso en la industria de la televisión por cable y el equipo Green llegó a ser conocido como FirstPerson, desafortunadamente no pudieron encontrar un mercado para *7 y es cuando decidieron involucrarse en el mundo de la Internet a través de un navegador propio llamado WebRunner que, a diferencia del resto, permitía crear objetos animados en movimiento y contenido ejecutable mediante pequeños programas conocidos como —applets— que pueden ir por la Red sin importar en qué tipo de hardware terminen.
En enero de 1995, Oak pasó a llamarse Java (una variedad de café) y WebRunner se renombro a HotJava. Más tarde, el 23 de mayo, Sun anunció formalmente a Java en la conferencia de SunWorld.[3]
En enero de 2010 Oracle Corporación adquirió Sun Microsystems y actualmente (2024) Java es una marca registrada de Oracle.

1.1.2. Versión

El listado de versiones Java SE y cada una tiene su pequeña historia, un conjunto establecido de conocimientos, una que otra característica y algunas diferencias (subjetivas) con su predecesora. Aquí las referencias son muy importantes, pues listar todas las características consumiría muchas hojas, estas son las que apuntan al documento (libro o website) que contiene la información.

JDK 1.0a En diciembre de 1994, se publicaron el código binario de Java y HotJava (en este momento todavía se llama Oak) en un archivo secreto en las profundidades de la Red; sólo un selecto grupo de amigos y una pequeña red informal de desarrolladores fueron invitados a probarlo. En sí sólo habían siete u ocho copias binarias fuera de Sun.[4][3]

JDK 1.0a2 ("full public") El *23 de mayo de 1995* la gente de Sun anunció que la tecnología Java era real y oficial.[3]

JDK 1.0 *23 de enero de 1996*. La versión de arranque [5, p. 53], tenía 12 packages, 212 clases e interfaces pero aún no estaba listo para desarrollar aplicaciones serias.[6][7][8]

JDK 1.1 *febrero de 1997*. Mejoraron el manejo de eventos AWT, clases internas, JavaBeans, JDBC, RMI.

J2SE 1.2 *diciembre de 1998*. Se cambió de nombre a 'Java 2' para distinguir de las anteriores que ahora sus APIs se dividían en tres plataformas: Java 2 Standard Edition (J2SE), Java 2 Micro Edition (J2ME) y Java 2 Enterprise Edition (J2EE). A partir de ahora el nombre 'JDK' representase específicamente a 'J2SE'.

J2SE 1.3 *mayo de 2000*

J2SE 1.4 *Febrero de 2002*. Lista completa de cambios [9].

Nueva palabra reservada `assert`→P.74.

J2SE 5.0 *Septiembre de 2004*

Java SE 6 *Diciembre 2006*. A partir de esta versión, el nombre perdió el '2' y el 'punto cero' porque Java es una marca registrada y abreviaturas como J2SE, J2ME, J2EE se convierten en Java SE, Java ME y Java EE (www.oracle.com/java/technologies/javase/javanaming.html).

Java SE 7 *Julio 2011*

Java SE 8 (LTS) *marzo de 2014*. Mejora en el lenguaje con un nuevo operador funcional, `->`→P.46 y referencia, `::`→P.60.

Tiene 217 paquetes.

A partir de JDK 8, March 18, 2014 se introduce en Java el paradigma de la programación funcional. Para ello se introduce las interfaces funcionales (Lambdas), `Optional`→P.89 y `Stream`→P.253.

Java SE 9 *Septiembre 2017*

Java SE 10 (18.3) *Marzo 2018*

Java SE 11 (18.9 LTS) *Septiembre 2018*.

Tiene 59 modulos.

Java SE 12 (19.3) *marzo de 2019*. Oracle modificó sus acuerdos de licencia, requiriendo que algunos tipos de usuarios paguen una tarifa de suscripción para utilizar la versión de Java de Oracle.

Java SE 13 *Septiembre 2019*

Java SE 14 *Marzo de 2020*. Mejora en el lenguaje con `switch`→P.67 *expressions*; e introduce una nueva palabra clave: `yield`→P.68.

Java SE 15 *Septiembre 2020*. Mejora en el lenguaje con un nuevo literal String: *TextBlocks*, `"""`→P.155.

Java SE 16 *Marzo 2021* conforme JSR 391. Funciones nuevas [10], que incluyen: Mejora en el lenguaje, introduce una nueva palabra clave: `record`→P.145; nueva caracteristica (tipo de dato): ⟨*TypePattern*⟩→P.39 para `instanceof`→P.47 Pattern Matching.

Java SE 17 (LTS) *Septiembre 2021* conforme JSR 392. Nuevas características [11]: Mejora en el lenguaje, introduce nuevas palabras clave: `sealed`→P.108, `non-sealed`→P.109, y `permits`→P.127.

Java SE 18 *Marzo 2022* conforme JSR 393.

Java SE 19 *Septiembre 2022* conforme JSR 394.

Java SE 20 *Marzo 2023* conforme JSR 395.

Java SE 21 (LTS) *Septiembre 2023* conforme JSR 396. Mejora en el lenguaje, `case`→P.69 Pattern Matching; nueva caracteristica (tipo de dato): ⟨*Record Patterns*⟩→P.40.

Java SE 22 *Marzo 2024* conforme JSR 397. Mejora en el lenguaje, introduce nueva palabra clave: `_`→P.38,

Java SE 23 *Septiembre 2024* conforme JSR 398 [12]. Mejora en el lenguaje con *comentarios de documentación en Markdown* de la forma: `///`→P.373.

Java SE 24 *Marzo 2025* conforme JSR 399 [13]. Nuevas caracteristicas: Class-File API (JEP 484); Stream Gatherers (JEP 485).

1.1.3. Cualidades

- Es un lenguaje orientado a objetos.
- Soporta la programación funcional; sin embargo, su naturaleza orientada a objetos, es todavía la principal forma de organización de código.
- Soporta diferentes modificadores de acceso que protegen los datos de accesos accidentales o malintencionados, lo que se conoce como *encapsulación*. Para muchos, la encapsulación es una característica particular de la programación orientada a objetos.
- Es de Arquitectura neutra, ya que los programas se compilan e interpretan en la Máquina Virtual Java (JVM) sin importar la arquitectura de la computadora subyacente.
- Es robusto, una de las mayores ventajas sobre C++ es que previene pérdidas de memoria, ya que es el propio lenguaje mediante el *Garbage Collector* quien se encarga de la gestión de memoria [14].
- Es sencillo, por lo menos más sencillo que C++ ya que elimina los punteros, la aritmética de punteros y la sobrecarga de operadores.
- Es seguro, porque ejecuta el código dentro de una máquina virtual (JVM). Lo que crea una sandbox o espacio virtual, es decir, un entorno de ejecución que aísla el código de posibles problemas.
- Es libre y de código abierto a través del proyecto OpenJDK

1.1.4. Plataforma

La plataforma Java® está compuesto por:

- Lenguaje
- Virtual Machine (VM)
- Bibliotecas/API

Java clasifica sus API's [15, p. 9] [16, p. 22] en tres plataformas que cubren distintos entornos de aplicación.

Java Platform, Standard Edition (Java SE)

Compone todas las bibliotecas centrales y necesarias para el desarrollo de aplicaciones de escritorio;

Capítulo 2

Variables

2.1. Introducción

2.1.1. Terminología

Los términos *variable*, *tipo* y *valor* se usan según lo definido por la Especificación del Lenguaje Java 8 y 24 (JLS 8, [23, §4.12, p. 80] y JLS 24, [13, §4.12, p. 96]).

2.1.2. Local-Variable Type inference (Java SE 10)

⟨*modificador*⟩ `var` ⟨*named*⟩ `=` ⟨*initialized*⟩`;`

Desde Java SE 10 (JEP 286) [24, §14.4, p. 447] puede evitar declarar el tipo explícito, poniendo en su lugar la palabra reservada `var`. Java infiere el tipo automáticamente del valor asignado en la expresión ⟨*initialized*⟩. Sin embargo, se debe tener las siguientes consideraciones:

- Es válido en un ámbito local, por ejemplo, dentro de un bloque o método.
- No se puede usar en una variable sin inicializar (asignación ⟨*initialized*⟩)
- La declaración y asignación deben ser en la misma línea para que pueda inferir el tipo.
- Una vez asignado el tipo de dato ya no se le puede asignar datos que no sean de este tipo, con excepción de `null`.
- No es posible inicializarlo en `null` porque no podrá inferir el tipo.
- No permite declaración compuesta (“`'var' is not allowed in a compound declaration`”).

```
var x = "Lorem ipsum";
System.out.println(x.getClass());
```

```
class java.lang.String
```

```
var x = "Lorem ipsum";
x = null;
//x = 234; // No compila, int cannot be converted to String

//var y = null; // No compila, cannot infer type for local variable y
```

```
int z; z = 123;
//var z; z = 234; // No compila, cannot infer type for local variable z
                  //  (cannot use 'var' on variable without initializer)

int u = 4, v = 45;
//var u = 4, v = 45; // No compila, 'var' is not allowed in a compound declaration
```

2.1.3. Tipos primitivos

⟨*modifiers*⟩ `boolean` ⟨*declarator*⟩`,` ⟨*additional declarators*⟩

El tipo booleano sólo admite valores `true`[→P.31] o `false`[→P.31]. Clase envolvente `Boolean`[→P.165].

```
boolean x = 4 > 5;
System.out.println(x);
```

```
false
```

⟨*modificador*⟩ `char` ⟨*declarator*⟩`,` ⟨*additional declarators*⟩

Valor entero, de 16 bits, entre 0 a 65535. Caracteres, con base el estándar Unicode [26, §1.3], desde U+0000 (NUL) hasta U+FFFF. Clase envolvente `Character`[→P.165].

```
char x = 64;
System.out.println(x);
```

```
@
```

⟨*modificador*⟩ `byte` ⟨*declarator*⟩`,` ⟨*additional declarators*⟩

Valor entero, de 8 bits, entre -128 y 127. Clase envolvente `Byte`[→P.166].

El tipo `byte` tiene 8 bits de longitud, por lo que su valor positivo máximo es 127, que es el número binario 1111111 (7 bits) con 1 bit reservado para el signo (+ o -).

```
byte x = 127;
```

⟨*modificador*⟩ `short` ⟨*declarator*⟩`,` ⟨*additional declarators*⟩

Valor entero, de 16 bits, entre -32768 y 32767. Clase envolvente `Short`[→P.166].

```
short x = 32767;
```

⟨*modificador*⟩ `int` ⟨*declarator*⟩`,` ⟨*additional declarators*⟩

Valor entero, de 32 bits, entre -2147483648 y 2147483647. Clase envolvente `Integer`[→P.166].

```
int x = 2147483;
```

⟨*modificador*⟩ `long` ⟨*declarator*⟩`,` ⟨*additional declarators*⟩

Valor entero, de 64 bits, entre -9223372036854775808 y 9223372036854775807. Clase envolvente `Long`[→P.167].

```
long x = 92248L;
```

Capítulo 3

Operadores

3.1. Introducción

3.1.1. Asignación

$x = y$ (infix operator)

Este operador evalúa y, luego asigna el resultado en la variable x. En variables locales sirve para la *inicialización de la variable*; en variables de instancia sirve para especificar la *inicialización por defecto* (§6.1.5→P.96).

La inicialización de variable es establecer un valor inicial [15, p. 37] y para la variable x.

Si x es una variable local no inicializada, y la emplea en una expresión, se produce un error en tiempo de compilación (“`variable x might not have been initialized`”).

```
int x;
// System.out.println(x); // No compila: variable x might not have been initialized
x = 4;
System.out.println(x);
```

```
4
```

3.1.2. Aritméticos

$x + y$ (infix operator)

Este operador según el tipo de los operandos x y y, procede:

- si x y y son de tipo numérico, suma x a y;
- si x o y es de tipo `String`→P.149 (entonces se dice que esta en contexto de Strings [25, §5.4, p. 137], por tanto realiza la conversión [25, §5.1.11, p. 129]) al operando que no es de tipo `String`, invoca su método `toString`→P.148 del objeto (o el de su clase envolvente), finalmente, hace una concatenación de los operandos mediante el método `concat`→P.151.

```
System.out.println(2 + 5.);
System.out.println(2 + '@');
```

```
7.0
66
```

```
System.out.println("a" + 2);
System.out.println(2 + "a");
```

```
a2
2a
```

x - y (infix operator)

Resta x a y.

x * y (infix operator)

Multiplica x por y.

x / y (infix operator)

División de tipos numéricos, $\frac{x}{y}$. Cuyo resultado varia según el caso:

- si x y y son enteros, la división produce un cociente entero. Cualquier parte fraccionaria en una división de enteros simplemente se descarta (es decir, se trunca); no ocurre un redondeo;
- si x o y es de punto flotante, la operación división retorna un valor flotante, conforme la *promoción de tipos* (§3.5.2).

```
System.out.println(7 / 3);
System.out.println(7 / 3.);
```

```
2
2.3333333333333335
```

x % y (infix operator)

Residuo de la división de x por y.

El operador residuo [25, §15.17.3, p. 673] algunas veces es llamado operador mod→P.184.

```
System.out.println(7 % 3);
System.out.println(-100 % 30);
```

```
1
-10
```

3.1.3. Unario de suma y resta

+x (prefix operator)

Unario de suma, donde x es de tipo numérico.

Capítulo 4

Estructuras de control

4.1. If

`if` (⟨*condición*⟩) ⟨*body*⟩

Si la ⟨*condición*⟩ es `true` ejecuta ⟨*body*⟩.

```
java.util.function.IntBinaryOperator f = (a, b) -> {
    int max = b;
    if (a > b) max = a;

    return max;
};

System.out.println(f.applyAsInt(4,3) + ", " +
                   f.applyAsInt(5,7));
```

```
4, 7
```

⟨*sentencia if*⟩ `else` ⟨*body*⟩

Si la condición de la ⟨*sentencia if*⟩ es `false` ejecuta ⟨*body*⟩.

```
java.util.function.IntBinaryOperator max = (a, b) -> {
    if (a > b)
        return a;
    else
        return b;
};

System.out.println(max.applyAsInt(4,3) + ", " +
                   max.applyAsInt(5,7));
```

```
4, 7
```

⟨*sentencia if*⟩ `else if` (⟨*condición*⟩) ⟨*body*⟩

Si la condición de la ⟨*sentencia if*⟩ es `false` y la ⟨*condición*⟩ es `true` ejecuta ⟨*body*⟩.

```
java.util.function.IntBinaryOperator compare = (a, b) -> {
    if (a > b)
        return 1;
    else if (a == b)
        return 0;
    else
        return -1;
};

System.out.println(compare.applyAsInt(4,3) + ", " +
                   compare.applyAsInt(3,4) + ", " +
                   compare.applyAsInt(3,3));
```

```
1, -1, 0
```

4.2. Switch

4.2.1. Switch (Sentencia)

`switch` (⟨*expresión*⟩) {⟨*selección*⟩ ⟨*selección*$_2$⟩ ...}

Sentencia switch [25, §14.11, p. 500] de selección multiple; transfiere el control (pasa) de una ⟨*selección*⟩ a otra que satisface la ⟨*expresión*⟩. Cada ⟨*selección*⟩ tiene el significado preciso siguiente:

- ⟨*selección*⟩ ⟶ ⟨*switch label*⟩ `:` ⟨*sentencias*⟩
- ⟨*switch label*⟩ ⟶ `case` ⟨*constante*⟩
  `default`

Una expresión ⟨*constante*⟩ puede ser: un entero literal de tipo `char`, `byte`, `short`, `int`; o (desde Java 7) un literal de `String`; o un objeto constante `enum`[→P.120].

La ⟨*selección*⟩ de un caso (`case`) ocurre cuando la ⟨*constante*⟩ satisface la ⟨*expresión*⟩, entonces ejecuta las ⟨*sentencias*⟩.

La rama de ⟨*selección*⟩ `default` es opcional, y sus ⟨*sentencias*⟩ se ejecutan cuando ninguno de los casos satisface la ⟨*expresión*⟩.

Para salir del `switch` use la instrucción `break`[→P.72]. Cuando un caso se propaga (i.e., no incluye la sentencia `break`), añada un comentario (por ejemplo el comentario "`este caso se propaga`") donde normalmente se encontraría la sentencia `break`.[18, p. 16]

```
java.util.stream.IntStream.of(2,5,6,9).forEach(x -> {
    switch (x) {
        case 2:
            System.out.print("2");
            break;
        case 5:
            System.out.print("5");
            /* este caso se propaga */
        case 6:
            System.out.print("6");
```

Capítulo 5

Excepciones

Este capítulo cubre las capacidades de Java® SE que se relacionan con el manejo de errores y excepciones. El capítulo esta integrado fundamentalmente en dos módulos: Lenguaje y API. Módulo Nivel Lenguaje: tratar las palabras clave (del lenguaje Java) que permiten control de excepciones. Módulo Nivel de API: tratar la tipología y la jerarquía de excepciones (la descripción de las clases de la API de Java) que extienden `Throwable`→P.81.

5.1. Estructuras de control

Las estructuras para control de excepciones (con base en [15, Table 9.1, p. 227] y [33, p. 441]) son:

- `throw`→P.77 —lanzar— para lanzar una excepción;
- `try`→P.78 —tratar— para tratar de ejecutar código que puede lanzar una excepción;
- `catch`→P.78 —captura— para gestión de excepciones;
- `finally`→P.79 —finalmente— para liberar recursos; y,
- `throws`→P.118 —lanza— para indicar que el método lanza una excepción.

5.1.1. Throw

`throw` ⟨*nueva excepción*⟩

Esta instrucción [25, p. 535], para indicar un problema en tiempo de ejecución, permite lanzar explícitamente una ⟨*nueva excepción*⟩ de tipo `RuntimeException`→P.84.

La declaración de creación de ⟨*nueva excepción*⟩ tiene el significado preciso siguiente:

- ⟨*nueva excepción*⟩ ⟶ ⟨*runtime instance creation*⟩
- ⟨*runtime instance creation*⟩ ⟶ `new` ⟨*runtime-exception type*⟩
- ⟨*runtime-exception type*⟩ ⟶ ⟨*runtime-exception*⟩ | ⟨*? extends runtime-exception*⟩
- ⟨*runtime-exception*⟩ ⟶ `RuntimeException`→P.84

⟨*nueva excepción*⟩: es una declaración un nuevo objeto, instancia de, excepción de tipo `RuntimeException`→P.84.

⟨*runtime-exception type*⟩: si no es de tipo `RuntimeException`→P.84 o una subclase de esta; se produce un error en tiempo de compilación. Por ejemplo, si en lugar de `UnsupportedOperationException` pone `Exception` se genera el error "`unreported exception Exception; must be caught or declared to be thrown`".

```
public class Ejemplo {
    public static void main(String... vars){
        System.out.println("Lorem");

        if (true)
            throw new UnsupportedOperationException("ipsum");

        System.out.println("dolor");
    }
}
```

```
Lorem
Exception in thread "main" java.lang.UnsupportedOperationException: ipsum
        at Ejemplo.main(Ejemplo.java:6)
```

5.1.2. Try-catch

`try` (⟨*resources*⟩) `{`⟨*body*⟩`}` ⟨*catches*⟩ ⟨*finally*⟩

Esta instrucción [25, p. 845] permite separar el código que puede lanzar una excepción ⟨*body*⟩ del código que trata la excepción ⟨*catches*⟩.

⟨*resources*⟩: son una lista de recursos separados por `;`.

Opcionalmente puede agregar: ⟨*catches*⟩ una o mas clausulas `catch`→P.78; ⟨*finally*⟩ una instrucción `finally`→P.79

El código de un bloque try se ejecuta normalmente y si alguna orden lanza una excepción entonces un bloque `catch` toma el control. Pero si el bloque try se ejecuta normalmente entonces el o los bloques catch se ignoran.

Un bloque try, nesesariamente debe ir seguido de una orden catch o finally o ambos.

```
try {
    int x = 1 / 0;
} catch (Throwable e) {
    System.out.println(e);
}
```

```
java.lang.ArithmeticException: / by zero
```

⟨*sentencia try*⟩ `catch` (⟨*excepción*⟩) `{`⟨*body*⟩`}`

Si el bloque de la ⟨*sentencia try*⟩ lanza una ⟨*excepción*⟩ ejecuta ⟨*body*⟩.

La clausula `catch` permite gestionar la ⟨*excepción*⟩ de `try`→P.78 [23, p. 718], [25, p. 845].

La declaración de variable del argumento ⟨*excepción*⟩ tiene el significado preciso siguiente:

- ⟨*excepción*⟩ ⟶ ⟨*modifier*⟩ ⟨*type*⟩ ⟨*named*⟩
- ⟨*type*⟩ ⟶ ⟨*throwable type*⟩ | ⟨*throwable type*⟩`|`⟨*throwable type*⟩
- ⟨*throwable type*⟩ ⟶ ⟨*throwable*⟩ | ⟨*? extends throwable*⟩
- ⟨*throwable*⟩ ⟶ `Throwable`→P.81
- ⟨*named*⟩ ⟶ ⟨*identificador*⟩ | ⟨*unnamed variable*⟩

Capítulo 6

Clases

6.1. Introducción

6.1.1. Terminología

Los términos *instancia*, *tipo*, *campo* (field [35, Def. 2.2.1, p. 17]) y *método* se usan según lo definido por reflexión (clase `Class`→P.329) y la JLS 22 [25, §8].

6.1.2. Declaración Clase

⟨*class modifiers*⟩ `class` ⟨*identifier*⟩⟨*type-parameters*⟩ ⟨*super-class*⟩ ⟨*super-interfaces*⟩ ⟨*permitted-subclasses*⟩ `{`⟨*body*⟩`}`

Declara [23, p. 193] la clase ⟨*identifier*⟩. Clase envolvente `Class`→P.329.

Una clase es una agrupación (⟨*body*⟩) de variables miembro (datos) y funciones miembro (métodos) que operan sobre dichos datos y permiten comunicarse con otras clases. Las clases son verdaderos tipos de variables o datos, creados por el usuario.

⟨*class modifiers*⟩: Modificadores de clase (véase `classModifiers`→P.336) por ejemplo, `public`→P.103, `sealed`→P.108, etc.

⟨*identifier*⟩: el identificador de la clase (conforme [18, §9, p. 18/22]) debe ser sustantivo. Cuando son compuestos tendrán la primera letra de cada palabra que lo forma en mayúsculas (a esto se denomina *Camel Case*). Intente mantener los nombres de las clases simples y descriptivos. Use palabras completas, evite acrónimos y abreviaturas (a no ser que la abreviatura sea mucho más conocida que el nombre completo, como URL o HTML).

⟨*type-parameters*⟩: parámetros de tipo, si es una clase genérica §11→P.301.

⟨*super-class*⟩: es la clase de la cual hereda, `extends`→P.121.

⟨*super-interfaces*⟩: las interfaces que implementa, `implements`→P.125.

⟨*permitted-subclasses*⟩: las clases que tiene permitido extender de esta, `permits`→P.127.

⟨*body*⟩: es la estructura del cuerpo de la clase y esta formada por los siguientes elementos:

1. la declaración de campos es opcional y pueden ir en cualquier parte de la clase (fuera de cualquier método);
2. la declaración de métodos es opcional y puede ir en cualquier parte de la clase. En Java NO puede haber métodos dentro de métodos;
3. la declaración de bloques *inicializadores* de clase es opcional.



En la declaración del ⟨*body*⟩ se recomienda [18, §6.4, p. 13/22] seguir las siguientes reglas de formato:

- la llave de apertura "{" aparece al final de la misma línea de la sentencia de declaración;
- la llave de cierre "}" empieza una nueva línea indentada ajustada a su sentencia de apertura correspondiente, excepto cuando no existen sentencias entre ambas, que debe aparecer inmediatamente después de la de apertura "{".

```
class Box {}

public class Ejemplo {
    public static void main(String... args) {
        Box box = new Box();
        System.out.println(box);

        Ejemplo ejemplo = new Ejemplo();
        System.out.println(ejemplo);
    }
}
```

```
Box@25359ed8
Ejemplo@21a947fe
```

Por default una clase tiene *miembros de clase* [25, §8.2, p. 259] heredados de `Object` (por ejemplo el método `toString`→P.148) otros de la ⟨*super-class*⟩ que hereda, de las ⟨*super-interfaces*⟩ que implementa, y, finalmente las declaradas en su ⟨*body*⟩.

```
class Box {}

public class Ejemplo {
    public static void main(String... args) {
        Box box = new Box();
        System.out.println(box.toString());
    }
}
```

```
Box@3d3fcdb0
```

6.1.3. Inicializadores de clase

⟨*class declaration*⟩ {... `static` {⟨*inicializadores de clase*⟩} ...}

Inicializadores de clase (`static`→P.105) [25, §8.7] se utilizar para dar valor a las variables estáticas [7] se ejecutan en orden de aparición una sola vez, cuando se crea el primer objeto de esta clase.

⟨*class declaration*⟩ {... {⟨*inicializadores de instancia*⟩} ...}

Inicializadores de instancia [25, §8.6] se llaman cada vez que se crea un objeto se ejecutan en orden de aparición

Capítulo 7

API fundamental

La Programación usando las API's de Java cubre las capacidades fuera del Lenguaje estandar. Al igual que la instrucción `using` o `\usepackage` para los lenguajes C#, y LaTeX 2_ε respectivamente, el uso de una dependencia en Java® se declara mediante `import`→P.97.

7.1. Introducción

7.1.1. El módulo `java.base`

Module `java.base`

Este módulo contiene los paquetes fundamentales de Java.

7.1.2. El paquete `java.lang`

Package `java.lang` (module `java.base`)

Este paquete (del modulo `java.base`) contiene las clases fundamentales de Java.

Por ejemplo, las clases `Object`→P.147, `System`→P.207, `Throwable`→P.81. son parte de este paquete.

7.1.3. La clase `Object`

Class `Object` (package `java.lang`)

Todas las clases en Java heredan de esta clase incluso los de tipo Array, por lo que todas heredan sus 11 métodos (algunos de los cuales están sobrecargados) [15, p. 279], [33, p. 352].

Constructor `Object()`

Constructor de la clase.

```
Object x = new Object();
System.out.println(x);
```

```
java.lang.Object@3d4eac69
```

Métodos de clase (11):

Method `Class<?> getClass()`

Retorna (en tiempo de ejecución) un objeto del tipo `Class`→P.329.

```
Class<?> x = new Object().getClass();
System.out.println(x);
```

```
class java.lang.Object
```

Method `protected Object clone() throws CloneNotSupportedException`

Crea y devuelve una copia de este objeto.

Method `boolean equals(Object obj)`

Retorna `(this == obj)`.

Method `protected void finalize() throws Throwable`

El Garbage Collector invoca al método `finalize` de una instancia, (siempre que hayan sido definidos por el programador de la clase) cuando el número de referencias al objeto sea cero.

Method `int hashCode()`

Método nativo, y este a su vez llama al método: `identityHashCode`→P.208 pasandole este objeto a su argumento. Retorna el código hash de este objeto.

```
Object x = new Object();
System.out.println(x.hashCode());
System.out.println(System.identityHashCode(x));
```

```
1028566121
1028566121
```

Method `void notify()`

Method `void notifyAll()`

Method `String toString()`

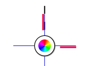
Retorna `getClass().getName()+ "@" + Integer.toHexString(hashCode())`.

Method `void wait() throws InterruptedException`

Ejecuta `wait(0)`.

Method `void wait(long timeout) throws InterruptedException`

Method `void wait(long timeout, int nanos) throws InterruptedException`



Capítulo 8

Stream API

8.1. Introducción

8.1.1. La API Stream (java.util.stream)

Package `java.util.stream` (module `java.base`)

La API Stream definidos en Java SE 8, se pueden clasificar en:

1. Interfaces (6)
   - `BaseStream`→P.266
   - `Stream`→P.253
   - `IntStream`→P.266
   - `LongStream`→P.267
   - `DoubleStream`→P.267
   - `Collector`→P.268
2. Classes
   - `Collectors`→P.262
   - `StreamSupport`→P.269

8.1.2. Stream interface

Interface `Stream<T>` (package `java.util.stream`)

Esta extiende de `BaseStream`→P.266`<T, Stream<T>>`

1. Métodos static (7)
   - `builder`→P.254
   - `empty`→P.254
   - `of`→P.254
   - `of`→P.254
   - `iterate`→P.254
   - `generate`→P.254
   - `concat`→P.254
2. Métodos de instancia (32)
   a) Operaciones intermedias (15)
      - `filter`→P.255
      - `distinct`→P.255
      - `sorted`→P.256
      - `sorted`→P.256
      - `peek`→P.256
      - `limit`→P.255
      - `skip`→P.255
      - `map`→P.256
      - `mapToInt`→P.256
      - `mapToLong`→P.256
      - `mapToDouble`→P.256
      - `flatMap`→P.257
      - `flatMapToInt`→P.257
      - `flatMapToLong`→P.257
      - `flatMapToDouble`→P.257
   b) Operaciónes terminales (17)
      - `toArray`→P.258
      - `toArray`→P.258
      - `forEach`→P.258

Method `static<T> Stream<T> of(T... values)`

Retorna `Arrays.stream(values)`.

```
// import java.util.stream.Stream;
Stream.of(1,2,3).forEach(System.out::println);
```

```
1
2
3
```

Method `static<T> Stream<T> of(T t)`

Retorna `StreamSupport.stream(new Streams.StreamBuilderImpl<>(t), false)`.

Method `static<T> Builder<T> builder()`

Retorna `new Streams.StreamBuilderImpl<>()`.

Method `static<T> Stream<T> empty()`

Retorna `StreamSupport.stream(Spliterators.<T>emptySpliterator(), false)`.

Method `static<T> Stream<T> generate(Supplier<T> s)`

```
Objects.requireNonNull(s);
return StreamSupport.stream(
        new StreamSpliterators.InfiniteSupplyingSpliterator.OfRef<>(Long.MAX_VALUE, s),
            false);
```

Method `static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b)`

```
Objects.requireNonNull(a);
Objects.requireNonNull(b);
@SuppressWarnings("unchecked")
Spliterator<T> split = new Streams.ConcatSpliterator.OfRef<>(
        (Spliterator<T>) a.spliterator(), (Spliterator<T>) b.spliterator());
Stream<T> stream = StreamSupport.stream(split, a.isParallel() || b.isParallel());
return stream.onClose(Streams.composedClose(a, b));
```

Method `static<T> Stream<T> iterate(final T seed, final UnaryOperator<T> f)`

```
Objects.requireNonNull(f);
final Iterator<T> iterator = new Iterator<T>() {
    @SuppressWarnings("unchecked")
```

Capítulo 9

Formateado

9.1. Introducción

Motivación en los métodos `System.out.printf` y `String::format` con el fin de dar una mayor especificación al formateo de datos pasado en sus argumentos.
`String`→P.149 con su método estático `format`→P.149 (Secuencia 9.1) y la clase `PrintStream`→P.209, con su método de instancia `printf`→P.210 que envía la salida con formato directamente al flujo de salida estándar, cuyo uso muy comun es en los campos estáticos out y err de la clase System.

Secuencia 9.1. Definición del método String::format

```
public static String format(String format, Object... args) {
    return new Formatter().format(format, args).toString();
}
```

9.2. Util

9.2.1. Formatter (java.util)

Class `Formatter` (package `java.util`)

Esta clase implementa `Closeable`, `Flushable` y es para presentación de los resultados.

Method `Formatter format(String format, Object... args)`

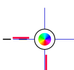
El método `format` realiza un formateado y para ello requiere de un *texto de formateo* y una *lista de argumentos* (que los interpreta como un array). El texto de formateo contiene *especificadores de formato* (Tabla 9.1) y estos hacen referencia a uno de los argumentos. Si hay más argumentos que especificadores de formato, los que sobran son ignorados.

Sobrecarga:

- `format(Locale l, String format, Object... args)`

```
import java.util.Formatter;
public class Ejemplo {
    public static void main(String[] args) {
        String usr = "Marco Antonio";
        int edad = 32;
```

PRINT PREVIEW (BETA TEST) ~ PRESENTACIÓN PRELIMINAR



```
        double prom = 78.26664;

        Formatter formatter = new Formatter();
        formatter.format("usr=%s, edad=%d, promedio=%.2f %%", usr, edad, prom);

        System.out.println(formatter.toString());
    }
}
```

```
usr=Marco Antonio, edad=32, promedio=78.27 %
```

9.2.2. Especificadores de formato

Tabla 9.1. Especificadores de formato

Conversion	Argument Category	Description
'b', 'B'	general	If the argument *arg* is `null`, then the result is "`false`". If *arg* is a `boolean` or `Boolean`, then the result is the string returned by `String.valueOf(arg)`. Otherwise, the result is "true".
'h', 'H'	general	If the argument *arg* is `null`, then the result is "`null`". Otherwise, the result is obtained by invoking `Integer.toHexString(arg.hashCode())`.
's', 'S'	general	If the argument *arg* is `null`, then the result is "`null`". If *arg* implements `Formattable`, then `arg.formatTo` is invoked. Otherwise, the result is obtained by invoking `arg.toString()`.
'c', 'C'	character	The result is a Unicode character
'd'	integral	The result is formatted as a decimal integer
'o'	integral	The result is formatted as an octal integer
'x', 'X'	integral	The result is formatted as a hexadecimal integer
'e', 'E'	floating point	The result is formatted as a decimal number in computerized scientific notation
'f'	floating point	The result is formatted as a decimal number
'g', 'G'	floating point	The result is formatted using computerized scientific notation or decimal format, depending on the precision and the value after rounding.
'a', 'A'	floating point	The result is formatted as a hexadecimal floating-point number with a significand and an exponent
't', 'T'	date/time	Prefix for date and time conversion characters. See Date/Time Conversions.
'%'	percent	The result is a literal '%' ('\u0025')
'n'	line separator	The result is the platform-specific line separator

Fuente:
file:///%LOCALAPPDATA%/Android/sdk/docs/reference/java/util/Formatter.html

Capítulo 10

Expresiones regulares

10.1. Introducción

10.1.1. Paquete java.util.regex

Package `java.util.regex` (module `java.base`)

Este paquete contiene las clases que permiten trabajar con expresiones regulares.

10.1.2. Entornos de desarrollo (Java API regex)

Métodos de String que trabajan con `regex` son: `replaceAll`→P.151, `matches`→P.151, `replaceFirst`→P.151, y `split`→P.152.

String::replaceAll Reemplazar la "o" por "0" siempre que sea precedida por una vocal.

```
String frase = "epopeya, la poesía heroica de los hombres.";
System.out.println(frase.replaceAll("o[aeiou]","0"));
//epopeya, la p0sía her0ca de los hombres.
```

Una expresión regular es una cadena de caracteres que representa un patrón (Pattern) de coincidencia (Matcher).

Por ejemplo: Es un nombre válido de una persona?

```
String patron = "[A-Z][a-zñáéíóú]+";
System.out.println("luis@75.pe".matches(patron)); //false
System.out.println("Martín".matches(patron)); //true
```

Ejemplo: Hay numeros hexa en esta oración?, de ser asi reemplace con la palabra que falta.[1]

```
String frase = "La noche de los 0xf1f2dfa se dispersó con la niebla...";
System.out.println(frase);
System.out.println(frase.replaceAll("0[xX][0-9a-fA-F]+", "dos"));
```

```
La noche de los 0xf1f2dfa se dispersó con la niebla...
La noche de los dos se dispersó con la niebla...
```

[1] La oración que sigue es un extracto de Alejandra Pizarnik (1955-1972).

10.1.3. Clase Pattern (java.util.regex)

Class Pattern (package java.util.regex)

Esta clase de constructor privado.

```
import java.util.regex.Pattern;

public class Ejemplo {
    public static void main(String... vars){
        String regex = "\\\\(?:[A-Za-z@]+|.)";
        Pattern p = Pattern.compile(regex);
        boolean x = p.matcher("\\macro").matches();
        System.out.println(x);

        boolean fox = Pattern.matches(regex, "\\macro");
        System.out.println(fox);
    }
}
```

```
true
true
```

10.1.4. Clase Matcher (java.util.regex)

Class Matcher (package java.util.regex)

Esta clase de constructor privado.

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Ejemplo {
    public static void main(String... vars){
        String regex = "\\\\(?:[A-Za-z@]+|.)";
        Pattern p = Pattern.compile(regex);

        Matcher m = p.matcher("\\macro");
        boolean x = m.matches();
        System.out.println(x);
    }
}
```

```
true
```

10.2. RegEx Language Specification

10.2.1. Especificación

Especificaciónes y la sintaxis de una **expresión regular**

Capítulo 11

Genéricos

<*x*> (group operators)

Type Arguments Or Diamond [23, p. 482].

Los *tipos genéricos* (*generic type*) fueron uno de los mayores cambios para la versión Java SE 5. Estos permiten que los *tipos* (clases e interfaces) sean parámetros al definir clases, interfaces y métodos. Al igual que los *parámetros formales* más familiares utilizados en las declaraciones de métodos, los parámetros de tipo proporcionan una forma de reutilizar el mismo código con diferentes entradas. La diferencia es que las entradas a los parámetros formales son valores, mientras que las entradas a los parámetros de tipo son tipos. Alguna de las ventajas del código genérico: Las comprobaciones de tipo son más estrictas en tiempo de compilación; Eliminación de `casts`.[1] Las colecciones fueron una de las API's más beneficiadas por ejemplo,

Secuencia 11.1. Sin genéricos

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

Secuencia 11.2. Empleando genéricos

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);   // no cast
```

11.1. Clase e interfaz genérica

Las clases que tenían que estar preparadas para colaborar con objetos de distinto tipo usaban `Object` para referirse a ellos.

```
public class Box{
    private Object object;

    public void set(Object object) {this.object = object;}
    public Object get() { return object; }
}
```

Dado que sus métodos aceptan o devuelven `Object`, usted es libre de pasar lo que quiera, siempre que no sea uno de los tipos primitivos.

[1] docs.oracle.com/javase/tutorial/java/generics/why.html

Capítulo 13

Reflexión

Package `java.lang.reflect` (module `java.base`)

API Reflexión.

13.1. Instancia de `Class` (`java.lang`)

Class `Class<T>` (package `java.lang`)

Las instancias de la clase `Class` representan clases e interfaces en una aplicación de Java corriente.[19]

El método `getClass()` de cualquier objeto, retorna (en tiempo de ejecución) un objeto del tipo `Class`. Dicho método es `final` y viene desde la clase `Object`, la madre de todo objeto. Una instancia de `Class` es un objeto que modela una clase en particular.

```
Object obj = "Lorem";
Class<?> x = obj.getClass();
System.out.println(x);
```

```
class java.lang.String
```

13.1.1. Variables de instancia y literal de clase

La forma de obtener un objeto tipo `Class` depende del contexto o las circuntancias.

1. **del objeto:** Tal como se vio en el ejemplo anterior, esta forma puede ser util si queremos consultar, por ejemplo quien lo instancio a dicho objeto (véase Subsección 13.1.2). Ademas permite delimitar mediante `extends`→P.121 la ventaja de este ultimo es que podemos instanciar un objeto directamente; aunque claro debe tener un constructor por default y por ello hay que gestionar la excepción.

```
String str = "hoy por la mañana";
Class<?> claseA = str.getClass();
Class<? extends String> claseB = str.getClass();
try{
    String strB = claseB.newInstance();
}catch(Exception ex){ex.printStackTrace();}
```

2. **Literal de clase:** Si queremos ver directamente una clase especifica y sabemos su nombre podemos llamarlo como *literal de clase* `.class`→P.32, esta forma es util, ademas ya viene con el argumento del tipo parametrizado correcto.